CISC 3115 EWQ6
Recursion and Recursive Math Functions

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Problem Solving using Recursion
- Recursive math functions
- Design solutions to recursive math functions using recursion
  - Define mathematical recursive function with base case
  - Design Java methods

Problem Solving using Recursion

- A divide-and-conquer problem solving approach where a problem can be divided into the same problems of smaller size
- Examples
  - Mathematical recursive functions
  - Sorting, searching
  - …

Mathematical Recursive Functions

- Such functions take their name from the process of recursion by which the value of a function is defined by the application of the same function applied to smaller arguments.
- Examples
  - Function to compute factorials
  - Function to compute Fibonacci numbers

Factorial

- Factorial of n is defined as
  - $f(n) = n! = n\ (n-1)\ (n-2)\ \ldots\ 1$
- whose recursive function can be
  - $f(n) = n\ f(n-1)$
- with the base case
  - $f(0) = 1$

The same problem of smaller size

The same function applied to smaller arguments

Computer Factorial

- Recursive function to compute factorial

$$f(n) = \begin{cases} nf(n-1) & if\ n > 0 \\ 1 & if\ n = 0 \end{cases}$$

- Example
  - f(4) = 4 f(3) = 4 3 f(2) = 4 3 2 f(1) = 4 3 2 1 f(0) = 4 3 2 1 1 = 24

Base Case

- Base case is important
- Otherwise, where do we stop (without the base case)? e.g., consider
  - f(3) = 3 f(2) = 3 2 f(1) = 3 2 1 f(0) = 3 21 0 f(-1) = 3 2 1 0 -1 f(-2) …
- The base case makes sure that we stop the recursive process somewhere.

Design Factorial Recursive Method

- Design: int factorial(int n)
- Observe:
  - Recursive function: f(n) = n * f(n-1) when n > 0
  - Bae case: f(0) = 1
- Design method factorial(n: int):
  - f(n) = n * f(n-1): when computing f(n), we invoke factorial(n) where we compute it by n * factorial(n-1), i.e., we invoke the same factorial method recursively.
  - f(0) = 1: we stop invoking the factorial method when n is 0.

Fibonacci Number

- Mathematical recursive function to compute Fibonacci numbers

$$f(n) = \begin{cases} f(n-1) + f(n-2) & if\ n > 1 \\ 1 & if\ n = 1 \\ 0 & if\ n = 0 \end{cases}$$

- What is the base case?

Design Fibonacci Recursive Method

- Design: int fibonacci(int n)
- fibonacci(n) is computed as
  - fibonacci(n-1)+fibonacci(n-2) when n>1 based on recursive function
  - $f(n) = f(n-1) + f(n-2)$ when $n>1$
- fibonacci(0) should return 0 and fibonacci(1) should return 1 according to the base case
  - $f(0) = 0$
  - $f(1) = 1$

Recursive Calls and Call Stack

- factorial(4) = 4 * factorial(3)

  = 4 * (3 * factorial(2))

  = 4 * (3 * (2 * factorial(1)))

  = 4 * (3 * (2 * (1 * factorial(0))))

  = 4 * (3 * (2 * (1 * 1))))

  = 4 * (3 * (2 * 1))

  = 4 * (3 * 2)

  = 4 * (6)

  = 24

- Observe the animation from the publisher and the author of the textbook (included below)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

= 4 * 3 * (2 * 1)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

= 4 * 3 * (2 * 1)

= 4 * 3 * 2

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1))))

= 4 * (3 * (2 * 1))

= 4 * (3 * 2)

= 4 * (6)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1))))

= 4 * (3 * (2 * 1))

= 4 * (3 * 2)

= 4 * (6)

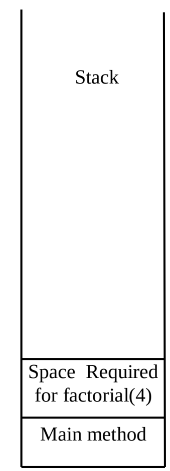
= 24

animation

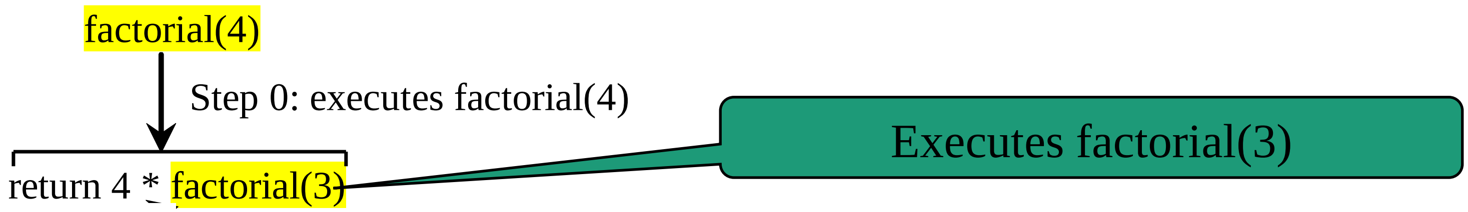
Trace Recursive factorial

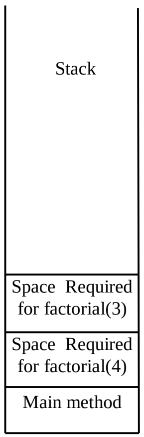
Executes factorial(4)

factorial(4)



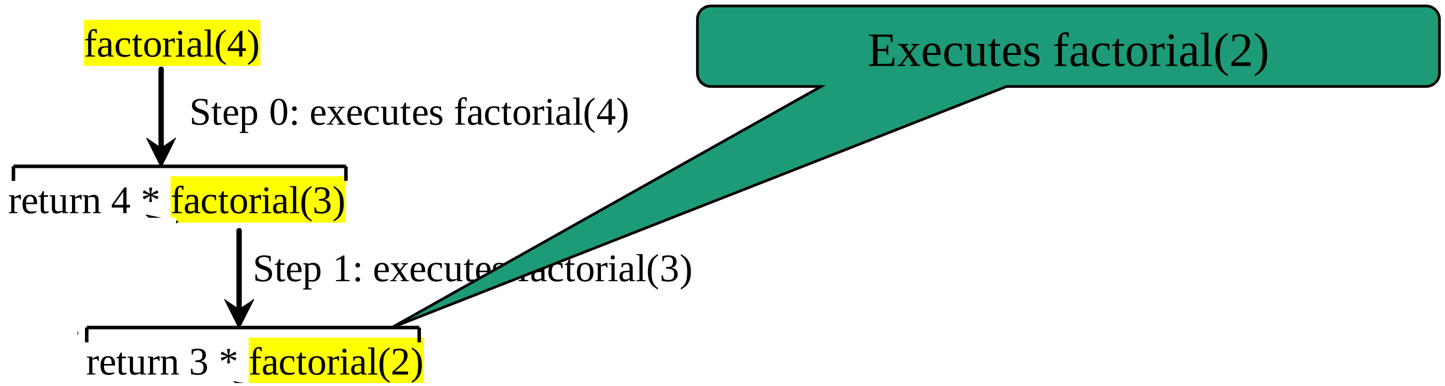
Trace Recursive factorial

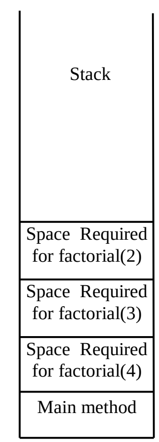




animation

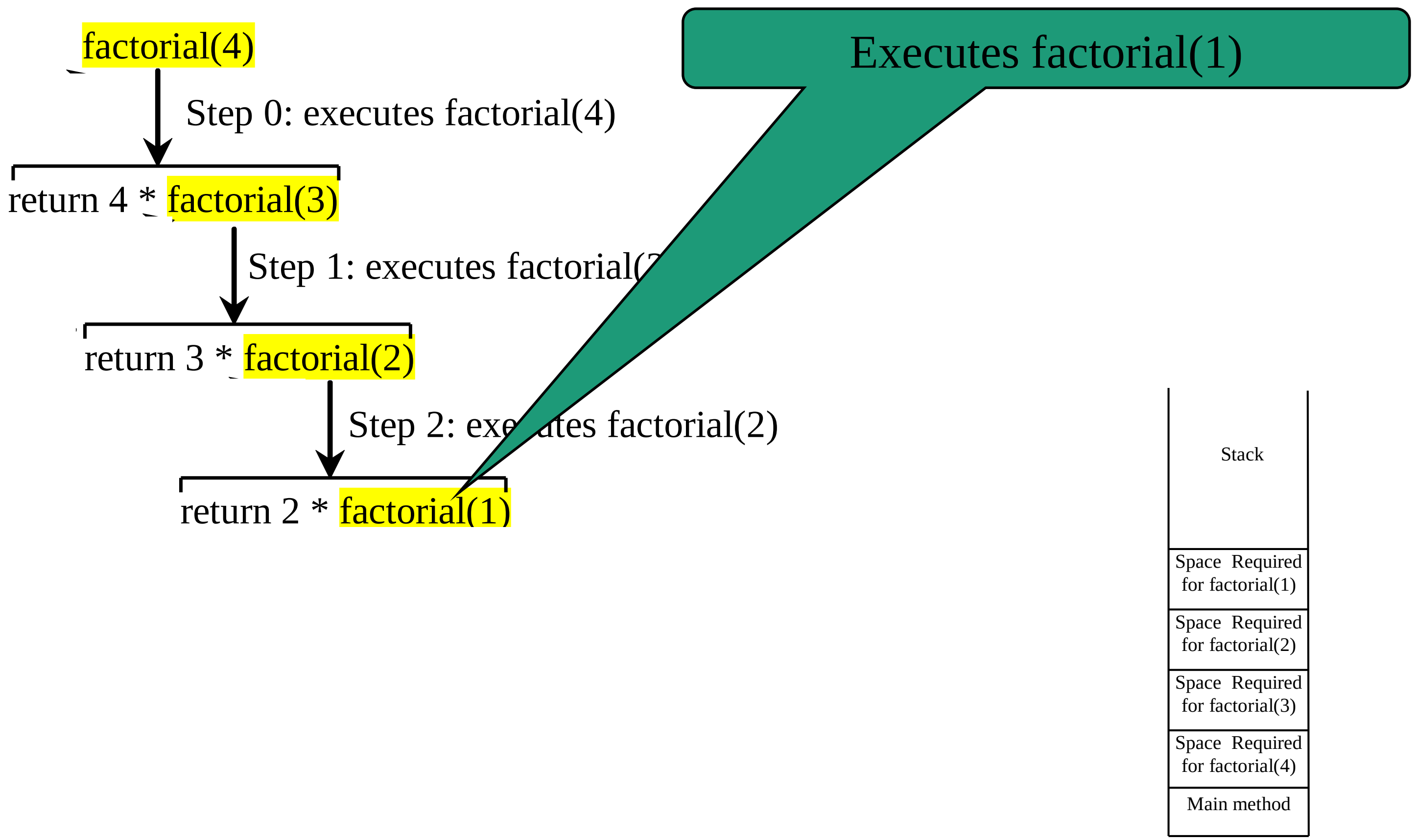
Trace Recursive factorial





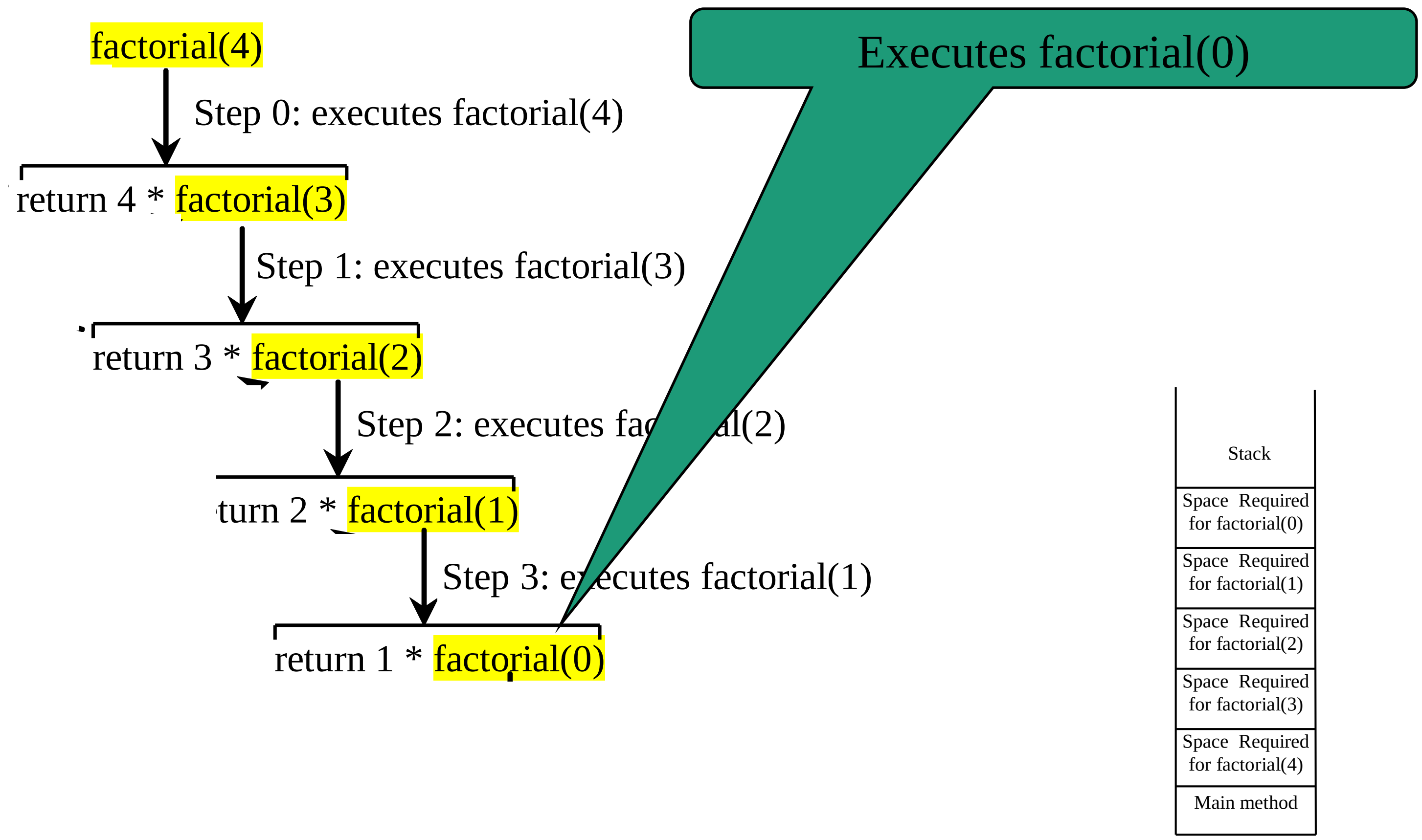
animation

Trace Recursive factorial



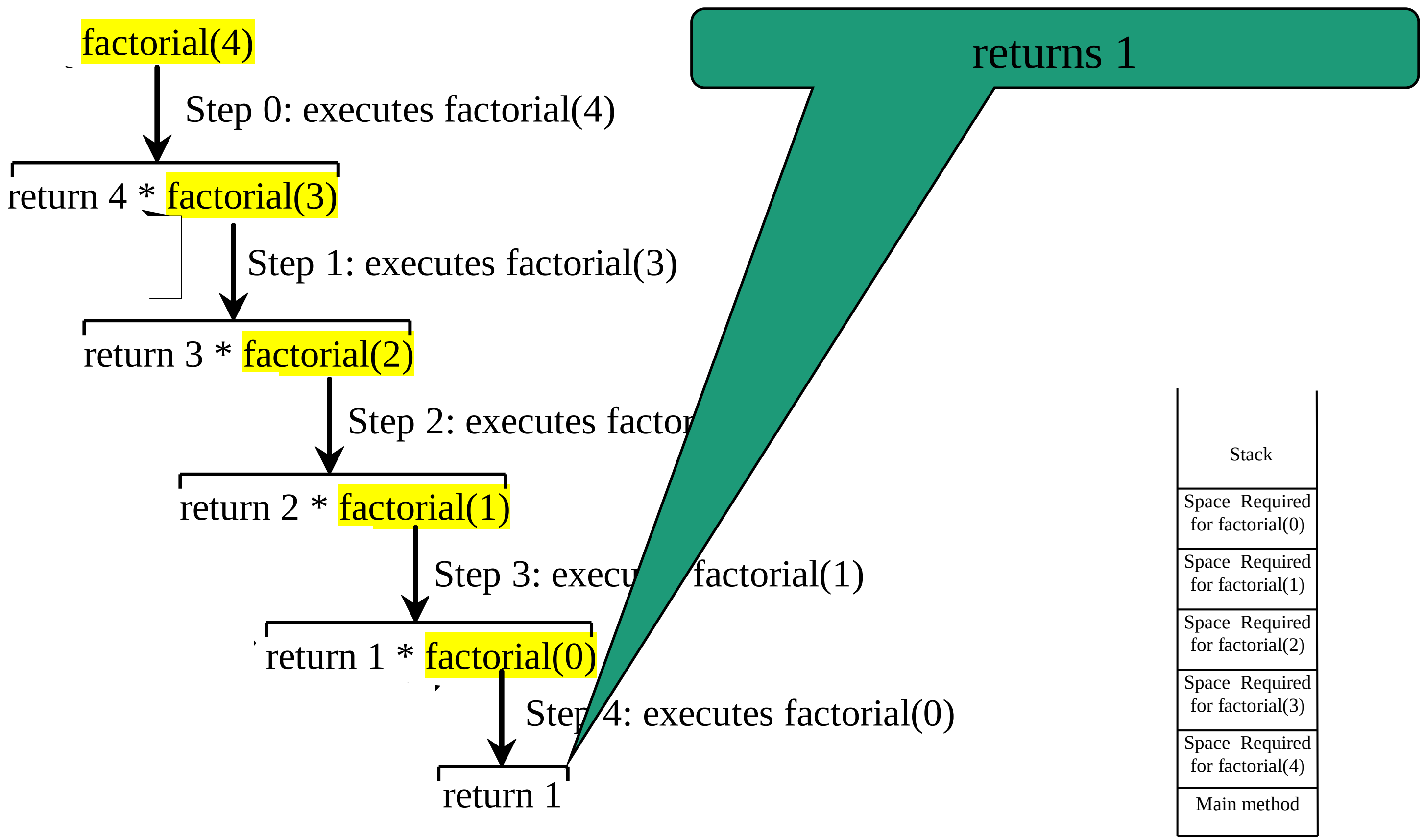
animation

Trace Recursive factorial



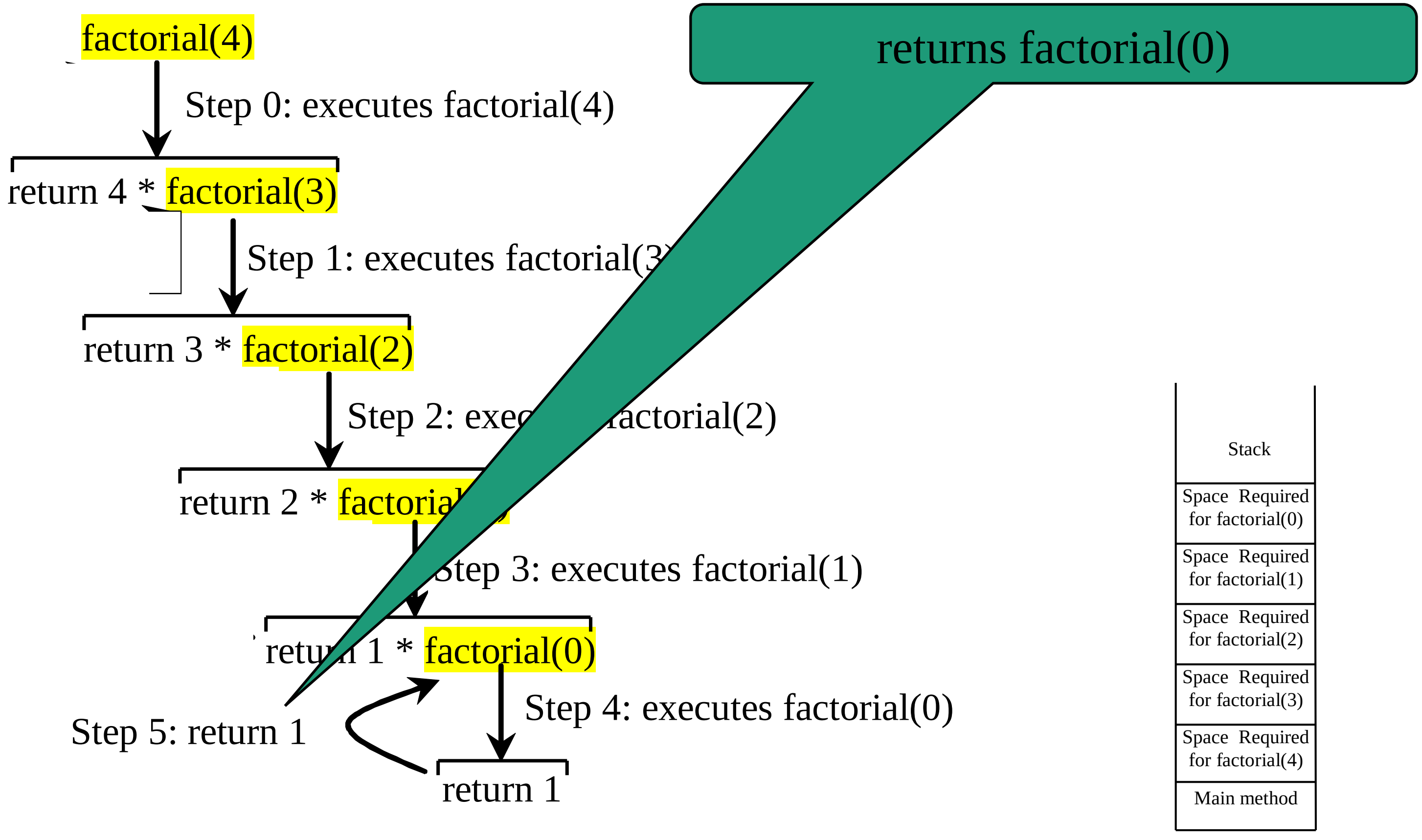
animation

Trace Recursive factorial



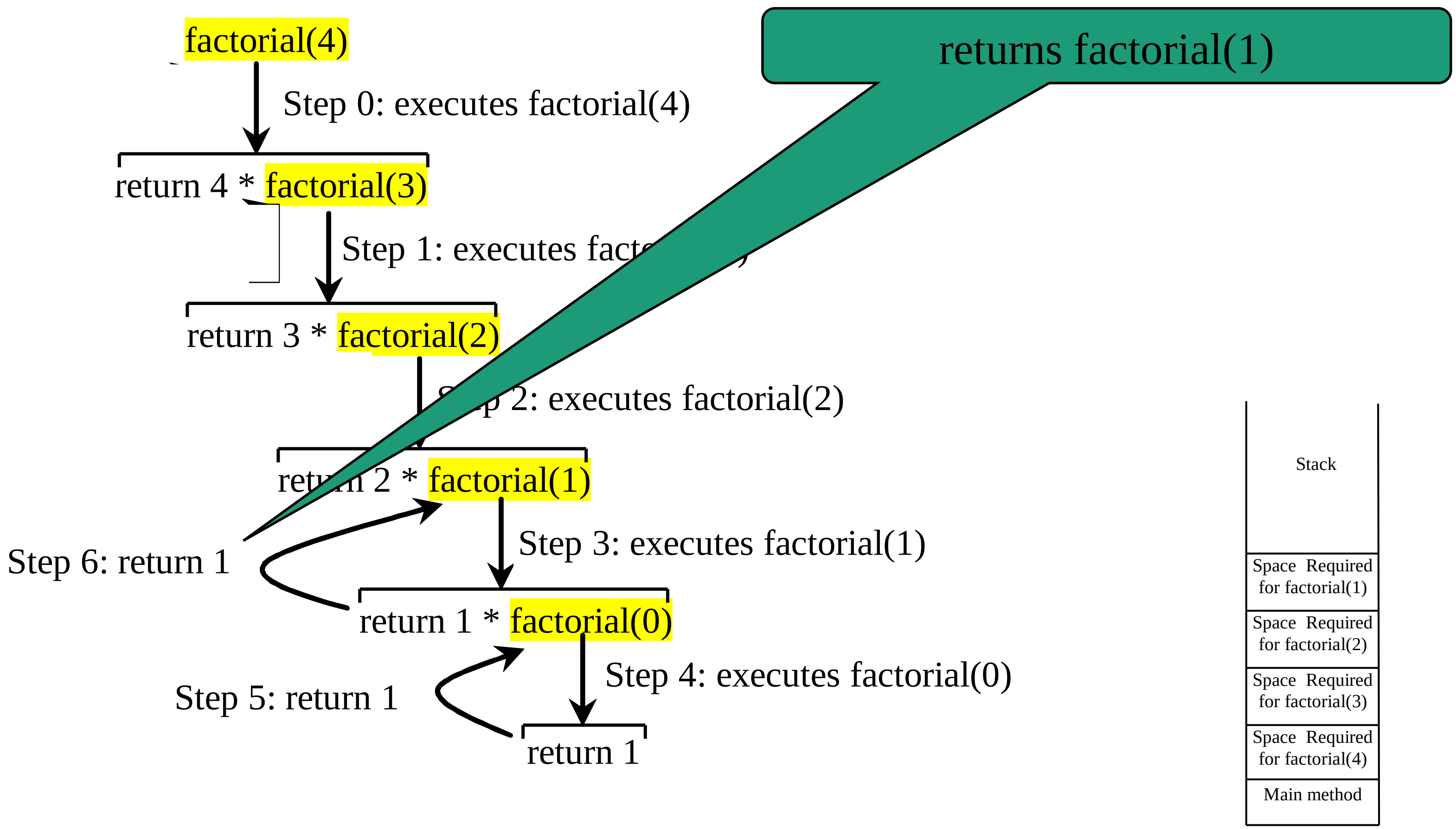
animation

Trace Recursive factorial



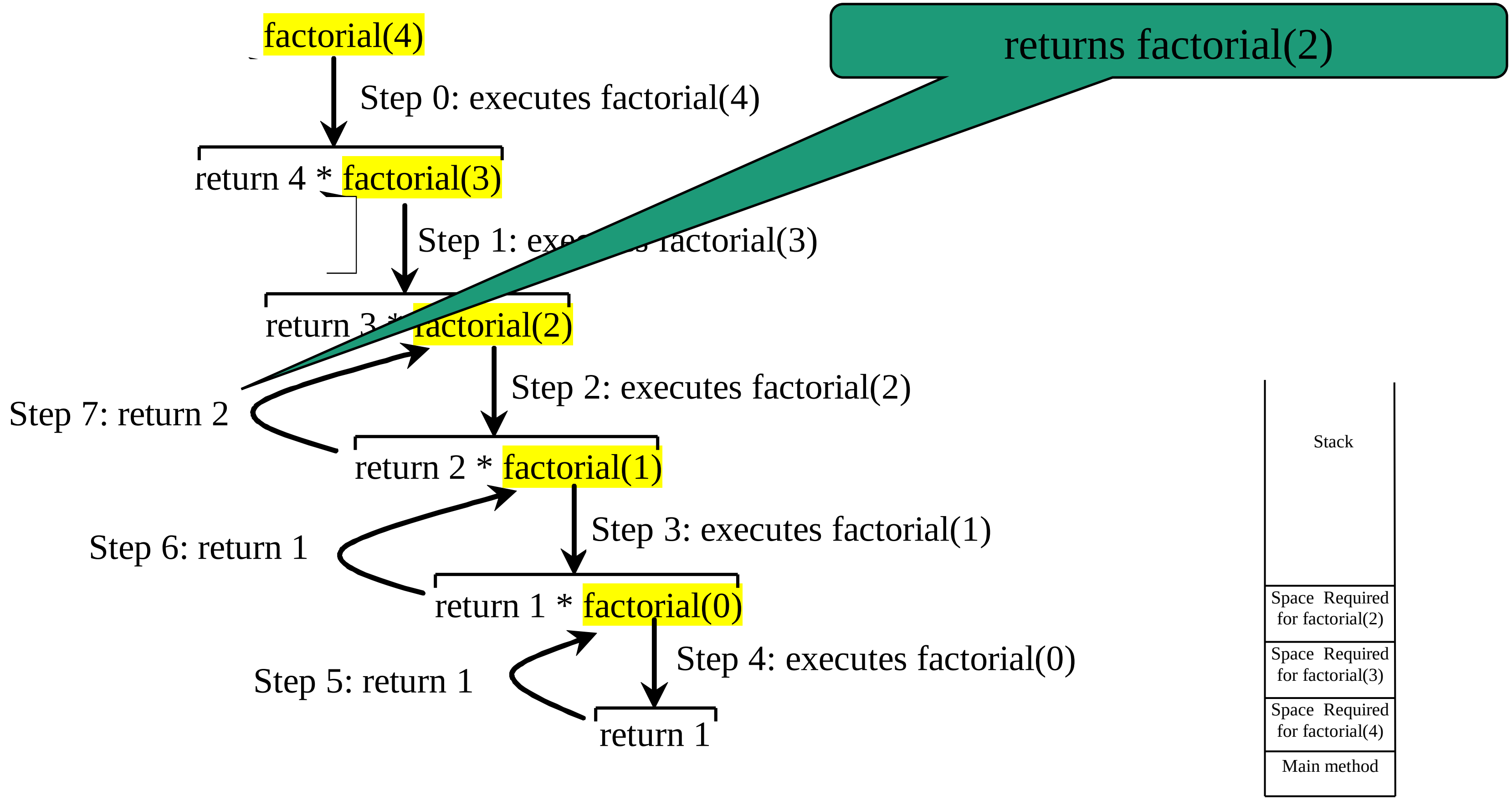
animation

Trace Recursive factorial



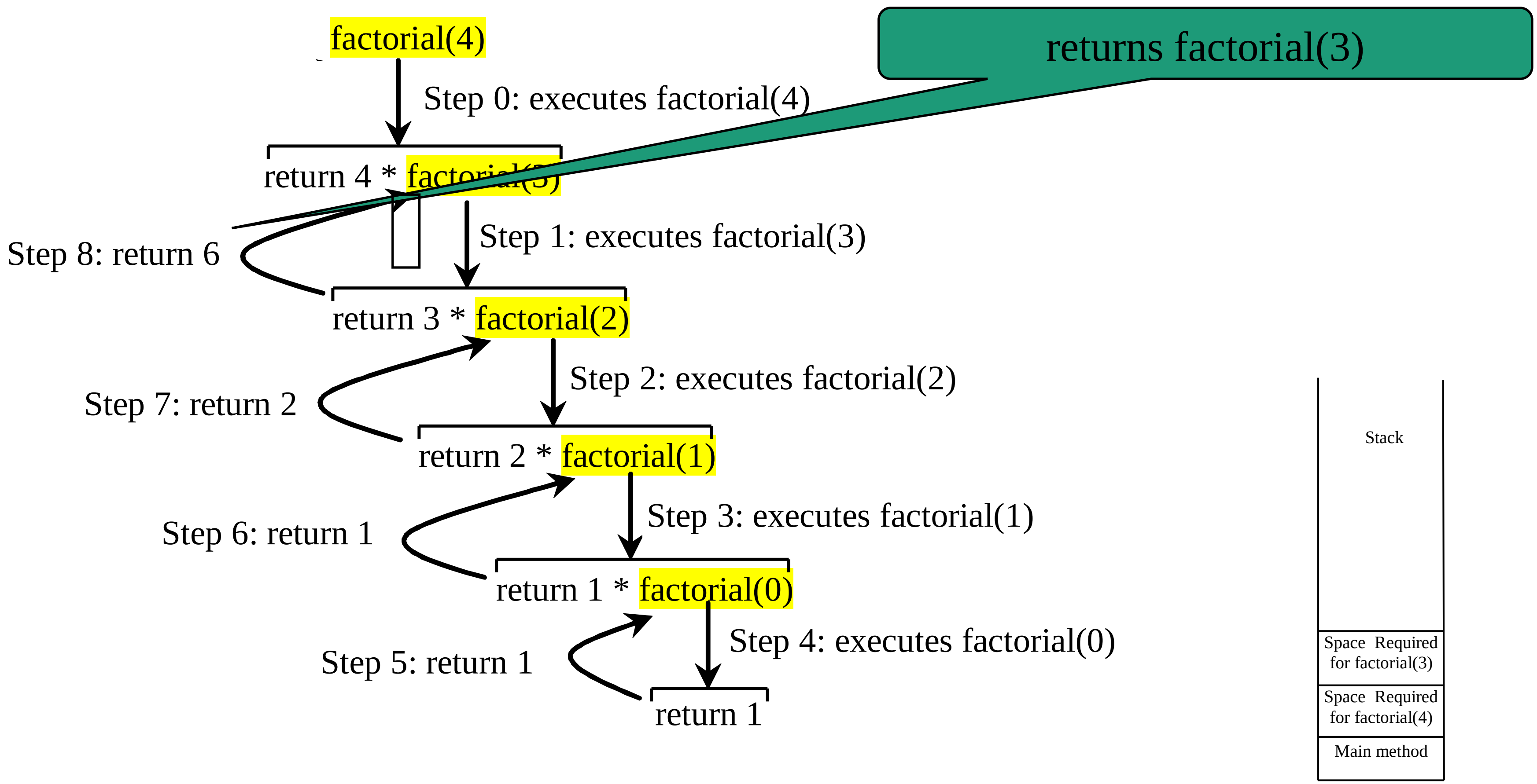
animation

Trace Recursive factorial

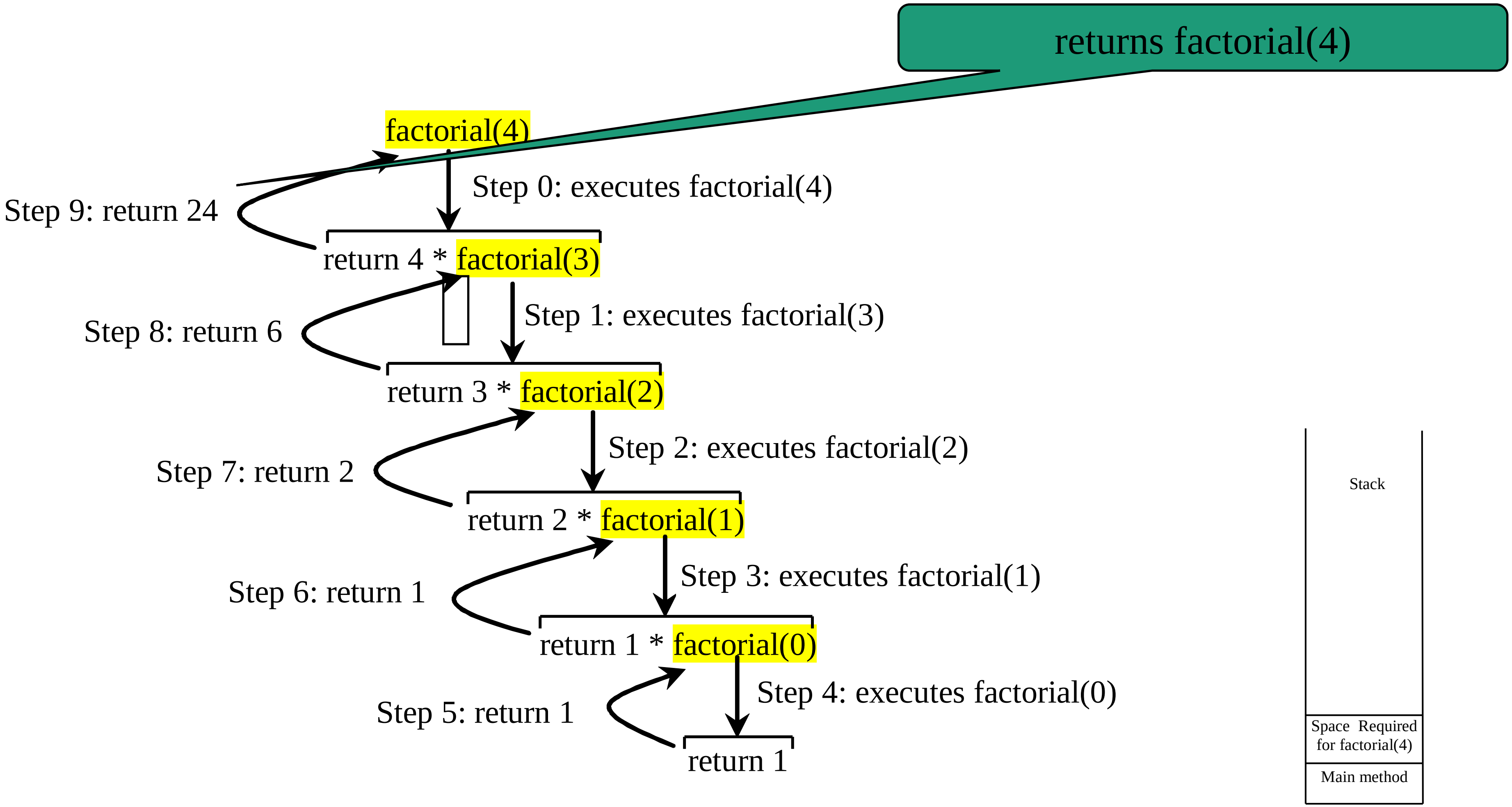


animation

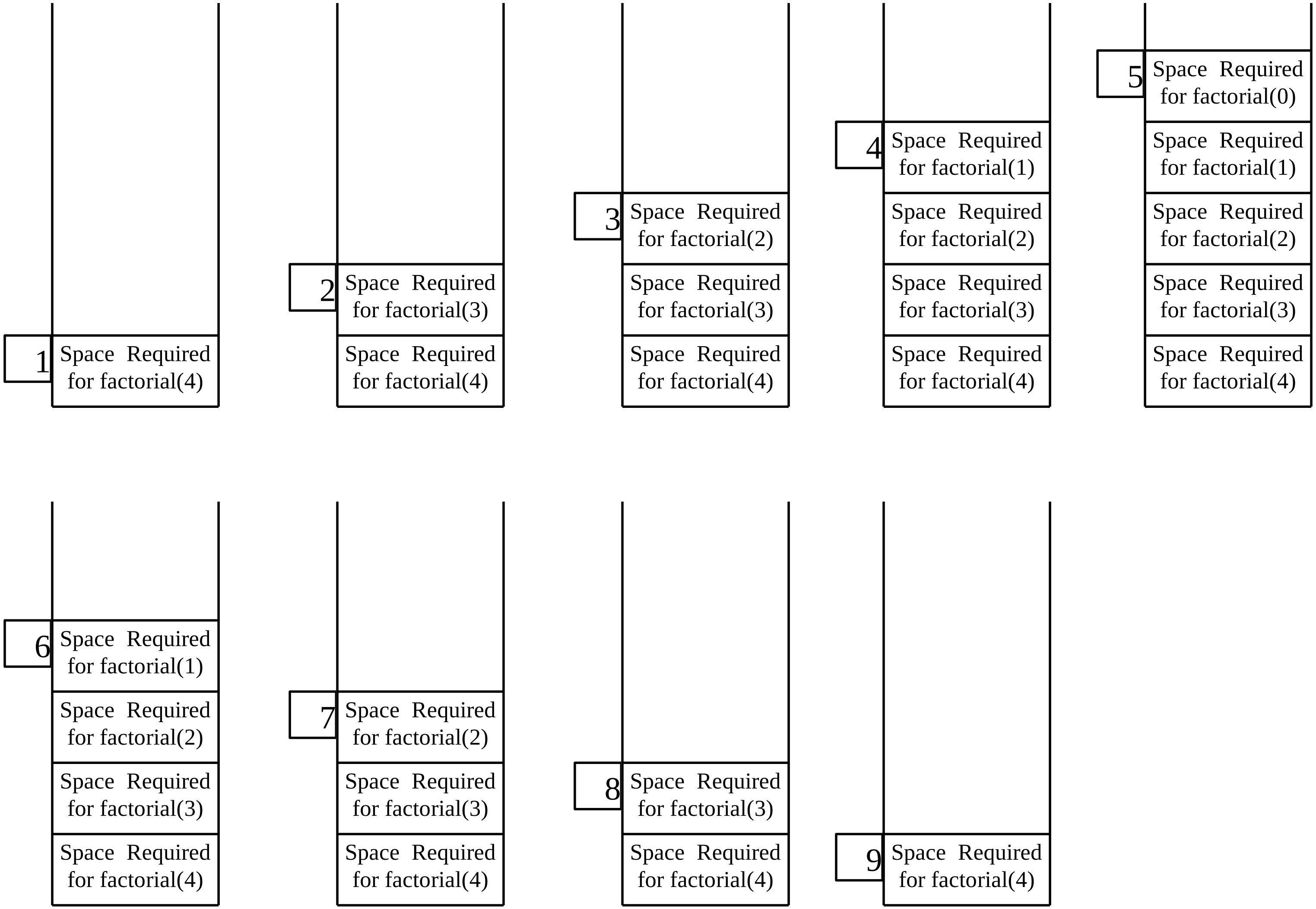
Trace Recursive factorial



Trace Recursive factorial



factorial(4) Stack Trace



Stack Overflow Error

- Neglecting or mishandling the base case will lead to a Stack Overflow error, for which, Java throws a StackOverflowError

```
$ java Factorial
Exception in thread "main" java.lang.StackOverflowError
        at Factorial.factorial(Factorial.java:3)
        at Factorial.factorial(Factorial.java:3)
        at Factorial.factorial(Factorial.java:3)
        at Factorial.factorial(Factorial.java:3)
...
```

Characteristics of Recursion

- All recursive methods have the following characteristics:
  - One or more base cases (the simplest case) are used to stop recursion.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Recursion as Problem Solving Strategy

- Break the problem into subproblems such that one or more subproblems resembles the original problem
  - These subproblems resembling the original problem is almost the same as the original problem in nature with a smaller size.
- Apply the same approach to solve the subproblem recursively to reach the base case

Questions?

- Concept of recursion
- Problem solving using recursion
  - Mathematical recursive functions
  - Base case
  - Call stack and stack trace
  - StackOverflowError